Mammoth: A Peer-to-Peer File System

Dmitry Brodsky, Jody Pomkoski, Shihao Gong, Alex Brodsky, Michael J. Feeley and Norman C. Hutchinson

Department of Computer Science
University of British Columbia

{dima,jodyp,shgong,abrodsky,feeley,norm}@cs.ubc.ca

Abstract

Peer-to-peer storage systems organize a collection of symmetric nodes to store data without the use of centralized data structures or operations. As a result, they can scale to many thousands of nodes and can be formed directly by clients. This paper describes the design and implementation Mammoth, which implements a traditional UNIX-like hierarchical file system in a peer-to-peer fashion. Each Mammoth node stores a potentially arbitrary collection of directories and files that may be replicated on multiple nodes. The only thing that links these nodes together is that each metadata object encodes the network addresses of the nodes that store it. Data is replicated by a background process whose operation is simplified by the fact that files are stored as journals of immutable versions. An optimistic replication technique is used to allow nodes to read and write whatever version of data they can access, while also ensuring consistency when nodes are connected. In the event of temporary failure, eventual consistency is achieved by ensuring that every replica of a directory or file metadata object receives all updates to the object, irrespective of delivery order. While an update is being propagated every node that receives it cooperates to ensure that the update is delivered, even if the original sender fails. Our prototype is implemented as a user-level NFS server. Its performance is comparable to a standard NFS server and it will be publicly available soon.

1 Introduction

At the core of a traditional distributed file system is a set of one or more tightly-coupled server nodes. These nodes coordinate client access to the file system they share, implementing key functions such as tracking where data is stored, maintaining consistency of cached or replicated data, and ensuring availability of important data. If multiple server nodes exist, the nodes must also ensure that they act as a tightly-coupled whole, for example, by handling node addition, removal, and failure in a coherent way.

Recent interest in peer-to-peer storage systems is motivated by the realization that this coordination and tight-coupling are an obstacle to scalability and to grass-roots information exchange. In a peer-to-peer system, by contrast, all nodes are symmetric and no service relies on central coordination. As a result, these systems can potentially scale to many thousands of nodes and they can be formed directly by groups of cooperating clients, without requiring the acquisition of a shared, centralized server. Implementing a peer-to-peer system, however, is challenging because it requires de-centralized alternatives to the functions typically performed by centralized server nodes.

Recent peer-to-peer work has demonstrated an intriguing idea that allows file data to be located using $O(\log N)$ messages sent among the N peer nodes that might store the item [6, 17, 18, 21]. The key idea is that each node is assigned a quasi-random ID and is linked to $O(\log N)$ other nodes chosen to ensure efficient lookup. Files (or blocks) are assigned an ID taken from the same space as the node ID and are stored on the node whose ID most closely matches that of the file (or block). Nodes are added by bootstrapping their neighbor lists from the existing node whose ID is closest to the new node, located in the same way that files (or blocks) are located. Finally, the key to tolerating failure is the random assignment of node IDs, which ensures that nodes with similar ID are unrelated to each other in the underlying network topology. Data replicated on nodes with similar IDs is thus reasonably invulnerable to a single point of failure.

The compelling simplicity of this approach comes with built-in limitations compared to a traditional file system. In particular, synchronization and consistency issues are avoided by making files immutable and by eliminating names and directories from the system. As a result, these systems have been described not as first-class file systems, but instead, as back-end archival repositories for use along with a more traditional file system. Clients are thus expected to continue to use a traditional file system for most of their work, referring to the peer-to-peer system only as necessary. Questions of how clients would coordinate sharing of files are not addressed by these systems.

Even in their role as an archive, these peer-to-peer systems require compromise. The main issue is their lack of flexibility with regard to where data is stored. The node that stores a file (or block) is determined when the file's identifier is assigned. This constraint allows the file to be located from any node efficiently, but it also has performance drawbacks. When a new node is added, for example, it immediately takes responsibility for all of its neighbor's files whose ID is now closer to the new node's ID, roughly half the files on that node. These files or forwarding proxies for them must be immediately copied to the new node. This lack of flexibility also complicates load balancing and caching, because storage location takes no advantage of access locality. Client-side caching, for example, is made impossible by the fact that clients with network proximity to each other follow largely disjoint paths to objects they share. Finally, while the $O(\log N)$ messages required to locate a file (or block) is small compared to N, the total expected latency to locate a file is $O(\log N)$ times the average access latency between two arbitrary nodes in the network, which for the Internet, for example, can be quite large.

This paper describes the design and implementation of an alternative peer-to-peer approach, which we have named *Mammoth*, that provides the full suite of features of traditional file systems while preserving the scalability and grass-roots sharing benefits of the peer-to-peer approach. Unlike previous peer-to-peer storage systems, our system implements a traditional file-system API, including directories and names. It is also differs in that it allows files and directories to be stored on any node and it adapts storage location dynamically to exploit locality, balance load, or ensure availability.

The key idea of our approach is to handle all coordination at the granularity of files instead of nodes. Each directory and file in the system is stored on at least one node and it may be replicated on other nodes for performance or availability reasons. In essence, the nodes that happen to store a particular file or directory act as the server for that file or directory: acting as location servers, maintaining the file's consistency and ensuring its availability. Nodes do not otherwise know about each other; and no operations require global knowledge of the network. Scalability is thus limited only by the number of nodes that store a particular file or directory and not by the total number of nodes in the network. This idea exploits the assumption that even in a very large system most files need be stored on only a relatively small set of nodes.

The remainder of this paper describes our design and implementation. This description begins with an overview of how nodes locate, access and modify directories and files. It then focuses on the protocol that nodes use to maintain consistency in the face of failures. We then discuss the administrative policy issues involved in ensuring that Mammoth performs well. Finally we describe some details of our publicly available prototype and we evaluate its performance.

2 Design overview

A Mammoth file system consists of a collection of peer-to-peer nodes that cooperate to store a UNIX-like hierarchical file system. Each file or directory is stored on one or more nodes, but no node stores everything. The system chooses an object's storage location adaptively to provide access locality and to ensure high availability. Replication occurs in background and is guided by flexible, user-controllable policies that allow different files to be replicated in different ways. Replica consistency is ensured using locks when sharing nodes are connected, but an optimistic approach is used when failures occur. Nodes are allowed to read and write whatever data they can currently access. Eventual consistency is simplified by storing directory and file metadata as logs and by storing file data as a collection of immutable versions. Write conflicts are stored directly in the metadata as history branches, leaving their ultimate resolution to higher-level software or to the user. Finally, security and access control issues are not addressed this paper.

This section describes these design features in more detail, focusing on the key data structures and operations, versioning and replication. Section 3 then describes how our design deals with failure.

2.1 Directory and file metadata

The key data structures that connect nodes to each other in Mammoth are directory and file metadata. Until a node stores a file or directory, it knows nothing about the rest of the file system (apart from its well-known root nodes) and the system knows nothing about the node. Adding a node simply involves storing a file or directory there and updating only that object's metadata to reflect the identity of this new storing node.

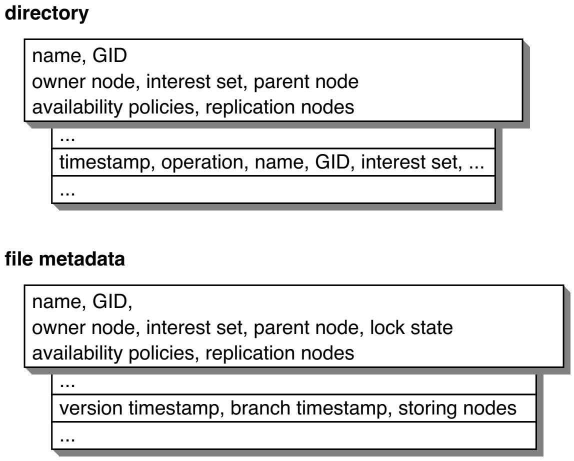

Figure 1: Directory and file metadata.

The format of Mammoth metadata is similar to that of a typical UNIX file system (e.g., directories and inodes), with three differences, as shown in Figure 1. First, each object's metadata lists the node or nodes that store it; nodes are named by their network address (e.g., IP address, port number pair). Second, the metadata is arranged as a timestamped change log, to simplify consistency. Third, file data, which is separate from metadata, is named symbolically and can be stored on a different node from the metadata.

We use the term *interest set* to refer to the set of nodes that store the metadata of a directory or a file. Multi-node interest sets are used to protect metadata from failure. One node in each interest set is chosen to act as the object's *owner* and thus synchronize updates to the object. For directories, nodes make changes via remote procedure call to the owner. For files, the system optimizes for update locality, by allowing nodes to make changes locally, using a multiple-reader-single-writer lock administered by the owner. Ownership moves to whatever node currently holds the write lock. Metadata changes are thus always made by the object's owner node, which is responsible for propagating these changes to the other nodes in the object's interest set. The algorithm described in Section 3 is used to ensure high availability and eventual consistency of metadata in the presence of failure.

To facilitate lookup, an object's interest set is also stored in its parent directory. Any file or directory can thus be located by starting at a node interested in the root directory and proceeding down the directory hierarchy to the target, sending messages to nodes interested in subdirectories as necessary. In the worst case, locating a file thus requires one message for each component of its pathname. Two additional data structures are used to optimize lookup. First, each object stores the address of one of the nodes interested in its parent directory as a hint. Second each node stores prefix tables that cache the results of recent lookups originated by the node.

The metadata itself is organized as a change log with each entry timestamped by the node that produced it to indicate the order in which it should be applied. This structure simplifies consistency, because it allows metadata updates to be applied in any order and still ensure eventual consistency. This property is particularly important to handle failures, as discussed in Section 3. Recall that in the absence of failure, the interest-set owner acts as a serialization point for all updates and can thus assign the update timestamp in a straightforward way.

Finally, unlike traditional UNIX directories — which map names to inode numbers — and inodes — which map offsets to disk blocks — Mammoth metadata names both symbolically. This approach allows a directory or file to refer to an entity that is stored on a remote node. These objects are named internally by a globally unique identifier (GID) comprised of their creator node's network address and a local timestamp. File data is named by a tuple comprised of the file's GID and the data's version timestamp.

2.2 File data

File data is stored as a journal of immutable file versions, as shown in Figure 1. A file's metadata logs information about these versions so that they can be located when needed. Each entry lists a version's timestamp, the timestamp of its history branch, and the network addresses of nodes that store it. Versions are timestamped by tuples consisting of the address of the node that created them and their creation time on that node. The history branch timestamp is explained in Section 3.3.

Mammoth's use of file versioning has two advantages. First, it confines consistency issues associated with replication to the file's metadata, because their immutability ensures that versions can be freely replicated without ever becoming inconsistent. Second, it allows the system to avoid replication of some versions as a performance optimization. In a typical replicated file system, by contrast, each update to a file is replicated using a consistency protocol that ensures that replicas see file operations in the same order. In Mammoth, on the other hand, only selected versions are replicated and they are replicated off the critical update path by a background process.

2.3 Replication

Finally, a central goal of Mammoth is to ensure high availability and to protect data from all forms of failure automatically, without requiring manual backup, for example. Our approach is motivated by the assumption that all files are not created equal and thus do not all require the same type of protection from failure. Further, we assume that, in general, users require protection from multiple failure modes that differ in their probability and also in the cost of protecting from the failure. For example, protection from node failure can be handled by replicating data to a nearby node. Protection from network failure or natural disaster, however, requires replication to a distant node that is unlikely to be affected by the failure.

Our solution is to support a set of replication policies for each of these modes. The policies have the general form of placing an upper bound on the number of hours of work that could be lost for a given failure. Users, administrators or the system assign policies to files based upon their type. The system then implements an object's replication policy using a background process that inspects recently modified directories or files to determine when they should be replicated and to what nodes. It then performs the necessary replication by copying the current version of the object to the appropriate other nodes.

3 Dealing with failure

Mammoth is designed to be robust in the face of intermittent node and network failure. It achieves this goal using an optimistic replication approach that allows nodes to read and write whatever version of data they can currently access even if other, even more current, replicas of that data exist, but are currently inaccessible. This section describes how the system permits this access and how it ensure that replicated data eventually becomes consistent when failed nodes or networks recover.

3.1 Access during partial failure

A node will first notice a partial failure that affects it when it tries to access an unavailable owner node. There are three situations in which this could occur: (1) when attempting to modify a directory, (2) when attempting to acquire a file read or write lock, or (3) when attempting to add itself to the interest set of a file or directory. Note that in the first two cases the node will already be in the interest set of the target object.

In any of these three cases, when a node discovers the owner is inaccessible, it immediately initiates an election procedure among the *interested* nodes that it can access to elect one of them to be the new owner. In the first two cases, it locally stores the addresses of all nodes in the interest set. In the third case, the target's parent directory stores this information and thus the election call is deferred to a node interested in the parent. Once a new owner is elected, the original access is allowed to complete. The existence of multiple owners and the resulting inconsistencies that arise are reconciled later when communication between the two owners is reestablished, as described in the next section.

Failure can also cause the current version of the file to be inaccessible. This version is always stored on the current owner, because creation of a new version requires a write lock and ownership always moves to the current write-lock holder. In addition, it is likely that the current version is replicated on other nodes. If the file was recently modified, however, it is possible that the most recently replicated version is somewhat older. In this case, if the file's owner is inaccessible, its current version may also be inaccessible. An accessing node must thus retrieve an older version of the file. It does this by consulting the version list stored in the file's metadata to determine the newest version that it can access. It scans this list in reverse chronological order examining the node lists of each version. Its scan is aided by a data structure maintained by each node that lists those nodes that are currently know to be down; this data structure is described in the next section. Nodes in the known-down list are skipped and others are contacted to request their version data. If the node is accessible, it responds with the requested version, and the requesting node's access completes with this version.

3.2 Metadata eventual consistency

File and directory metadata in Mammoth is stored only by nodes in the interest set of the object and is organized as an append-only log of timestamped metadata entries. The state of an object depends only on the entries in the log, and not on their order. As a result, the only way for metadata to be inconsistent is if some node has entries in its log that are missing from the log of another node.

When a node updates a file or directory, it attempts to update all nodes in the object's interest set. It enqueues any updates it is unable to send until communication with unreachable nodes is re-established. This procedure eventually reconciles partition inconsistencies as long as (1) a node does not permanently fail while holding enqueued updates and (2) a node has an accurate interest set for each object at the time it modifies it. The interest set may be inaccurate, however, if the interest set was modified in another partition prior to the update.

3.2.1 Incomplete update propagation

The first problem — failure of a node while it holds incompletely propagated updates — is resolved by requiring that nodes track updates they receive from other nodes. In essence, once a node receives an update from another node, it enters into a contract to take over propagation of the update should the original node fail. Update propagation is thus a two phase process: propagation of the update and propagation of a subsequent message that retires the update once all interested nodes are known to have received it. As an optimization, the updater can send additional messages to update the other nodes as it receives acknowledgments that nodes have received the update. In the meantime, each node maintains a table of the updates it has received, whose full propagation is unresolved. Each entry in the table stores a object's name, the address of the node responsible for propagating its update, and a list of suspect nodes (perhaps the entire interest set); other details of the update are stored naturally in the object's metadata.

This unresolved-update table is integrated with a liveliness module that provides an upcall in the event of the failure of a node responsible for propagating one of the updates in the table. This upcall initiates an election procedure among the nodes in the interest set to determine which of them should take over responsible for propagating the update. That node then plays the role of the original node and the remaining nodes return to a state of monitoring the liveliness of the newly elected node. The table is made persistent using lightweight transactions similar to RVM [20].

3.2.2 Inaccurate interest sets

The second problem — updating with an inaccurate interest set — is caused by our decision to favour availability over consistency, by allowing a node to become the owner of a file or directory whenever its current owner is unreachable. In a failure-free execution, an object's owner easily ensures that interest-set changes and other metadata updates are properly ordered. In the case of files, the owner ensures that a node's interest set is up-to-date before granting a write lock to it. In the case of directories, which are modified without locking, the owner adds a timestamp to its interest-set-change message so that receiving nodes can determine if they have made any updates ordered after the change, and if so, forward them to the newly interested node.

If an object does acquire multiple owners, we have a problem only if one of them adds a node to or removes a node from the object's interest set. If this happens, nodes in each partition will have different interest sets for the object, and thus metadata updates will not fully propagate to all interested nodes. This inconsistency will be easily detected, however, as soon one of the nodes in the intersection of the divergent interest sets becomes available in both partitions. When this happens that node will receive metadata updates from both owners and thus know to initiate reconciliation of the object. A problem remains, however, should partitioned interest sets diverge to the point that none of the nodes they have in common ever become available in both partitions. This situation only occurs, however, when all nodes in the intersection of the partitioned interest sets have *permanently* failed.

3.2.3 Permanent failure

The permanent failure of a node is detected when another node has enqueued updates for it that have been undeliverable for a long time. In this case, the node with the unpropagated update forcibly removes the failed node from the updated object's interest set and tells the other interested nodes to remove the update from their unresolved-update table. Any updates to this object made by the failed node that it was not able to propagated to at least one other node are now permanently lost.

Every forcible interest-set removal is also registered centrally to resolve the partitioned interest-set problem we just described. The table is replicated on a subset of the nodes in the root directory's interest set. It is indexed by a pair consisting of the object's GID and the address of the failed node. An entry stores an object's interest set at the time the failed node was forcible removed from it. If an interest-set partition exists, multiple nodes will register the same forcible removal, but with different interest sets. In this case, the registry updates the object's interest set to be the union of all registrations it has seen, thus joining the possibly disjoint interest sets. A registry entry can be removed when it is certain that no node will subsequently detect the permanent failure it names. Holding an entry for one full permanent-failure-timeout period is sufficient, because any node that takes more than this amount of time to detect the failure will have itself been deemed to have permanently failed.

3.3 File update conflicts

As in any optimistic replication scheme, update conflicts create inconsistencies in files or directories. In Mammoth these conflicts can occur in two situations: (1) when there are multiple owner nodes or (2) when the current version of a file is inaccessible and a node thus retrieves and modifies a recent, but not current, replicated version. The goal of Mammoth is to ensure eventual consistency of the update logs used to store directory and file metadata, but to leave resolution of conflicts in file data to higher-level software or to users. If such a conflict arises, it is stored as a branch in the version history of the directory or file. This history is visible to users, but a node that accesses a file or directory on a particular branch will by default continue to access that branch of the history. Users or application-level tools can inspect these histories and reconcile conflicts by merging branches.

A file version's immediate predecessor in normally determined by chronological timestamp order. If a conflicting update occurs during a period of disconnection and history branches are created, however, timestamp-order alone is insufficient to capture this relationship. In this case, the branch timestamp field in each version entry, which uniquely names the branch to which the version belongs, is used in conjunction with the version timestamp, as shown in Figure 1. New branch timestamps are assigned whenever a new owner must be elected due to failure.

As future work, we plan to examine the possibility of building a Bayou-like reconciliation facility as an application-level tool on top of Mammoth [22, 23]. To do this, Mammoth would be modified in two ways. First, applications would be provided with a facility for storing operation logs in parallel with files; these logs would then be used to resolve conflicts by merging the logs from two branches and replaying them to produce a version that contains updates from both branches, in the same way that Bayou does. Second, an upcall facility would be added to allow an application to register a conflict handler for a file or directory. Mammoth would use this facility to pass control to the tool when it creates a version-history branch.

3.4 Failure and replication

The long-term failure of a node that stores replicated data typically requires the re-replication of that data to preserve availability invariants established by the associated availability policies. In Mammoth this re-replication is handled by the nodes that store replicas.

Each node maintains summary information about the replicated data it stores. Included in this formation is a list of the other nodes that also replicate the same data. It informs the liveliness module of these nodes, requesting an upcall should any of them fail.

Should a replica node fail, this upcall initiates an election among the other nodes that share replicas with the node. This step is designed to ensure that as few nodes as possible proceed to the next step. In this next step, each node that has been elected selects a new replica node for the affected objects and sends its copies of these objects to the new replica node. Finally, the replica node sends message any node in the affected object's interest set informing it of the change in replication nodes. These messages are batched when possible to minimize message overhead. During this process the other replica nodes continue monitor this node and call a new election should this node fail.

3.5 Monitoring node liveliness

Mammoth nodes monitor other nodes for liveliness in three cases. As described in the prologue, they are: (1) when attempting to propagate an update to an interested node that appears to be down, (2) when holding an update that has not been fully propagated, and (3) when storing replicated data. In each case, the nodes in question are temporarily registered with the liveliness module along with an upcall procedure that the module triggers when the node transitions from down to up or up to down. The module determines liveliness by monitoring all in-bound and out-bound messages and sending ping messages to nodes when necessary.

4 Administrative policies

Mammoth's file-grain approach to coordination provides tremendous flexibility. It allows any number of replicated copies of any file or directory to be stored anywhere. As is typically the case, however, this flexibility can be used for good or bad. Furthermore, it is particularly important for a peer-to-peer system such as Mammoth to automatically administer itself, because its large scale and grass-roots nature argue strongly against active human administration. To use Mammoth's flexibility for good thus requires effective, adaptive policies that guide the decisions it makes. There are two principle administrative policies that fill this role.

4.1 Deciding where to replicate

The first important administrative policy is the one to decide where to place object and version-data replicas. There are two constraints.

The first constraint is that the replica nodes used by an object must satisfy its availability policies. This may require locating nodes, for example, that are in different buildings or are in completely different locations in the underlying network topology.

The second constraint is that recovering from the failure of a replica node should be efficient. Recall that replica nodes monitor each other to ensure that enough replicas are available. When a long-term failure is detected the nodes that share replicas with the failed node initiate a process to re-replicate these objects. The overhead of this process is determined by the total number of nodes affected in this way by the failure of a single replica node. In the worst case, every object on the failed node could be replicated on one distinct other node, thus requiring at least one message for every replicated object. If, on the other hand, the total number of affected nodes is small, the overhead will be minimal.

We solve this problem as follows. Each node maintains a database that describes a portion of the other nodes in the file system along with key characteristics important for availability policies (e.g., here the nodes are located, how they are connected to the network, who owns them, etc.). For a large system, this database will likely name only a subset of the nodes and it can be out of date. Nodes assign replica nodes to an object when they create it, when its availability policy changes or when a replica node fails.

To assign a replica set to an object, a node consults its local database to choose a single replica node. It chooses this node arbitrarily from among a set of candidate nodes that will satisfy the object's policy. The node then sends a request to this node requesting that it accept relicas for the object and that it choose the other nodes that will replicate the object. The replica node picks the other nodes by consulting both its local database, to ensure that the nodes it picks will satisfy the objects availability policy, and also the list it maintains that summarizes the other replicas it is already storing. This list includes the names of all other nodes with which it currently shares replicas; that is, the nodes that will be affected should it fail. It selects the additional replica nodes so that it keeps the number of nodes in this list small. If three or more replica nodes are required to satisfy a policy, the node consults these other nodes offering a list of acceptable candidates. These nodes select the candidates from this list that also tend to keep this number of nodes they share replicas with small. This process is simplified by the fact that objects can adequately be protected from failure by keeping only a few replicas, typically two or three.

4.2 Deciding when to cache metadata

The second important administrative policy is the one used to decide when to add a node to an interest set and when to remove one. Recall that nodes are placed in a directory or file interest set to exploit access locality. Performance of an object that is frequently accessed from a node is improved if the node is added to the object's interest set, because most accesses complete locally. In addition, performance is also improved if objects that are read frequently have large interest sets, because read load can be distributed among these nodes; this is the case for directories close to the root of a large Mammoth file system, for example. On the other hand, increasing the size of an object's interest increases the cost of updating the object, because updates are propagated to all interested nodes, and increases the costs of failure, because it increases the probability that some interested nodes will be down when the object is updated, thus requiring that the update be enqueued until the node is available.

We believe that it should be feasible to build an effective adaptive policy to strike a good balance between these concerns. We have thus far taken only a small step in this regard, setting access-frequency low- and high-water marks that the system uses to determine when to add or remove a node from an interest set. Access frequency statistics needed to implement this policy are collected by owner nodes.

5 The prototype implementation

This section describes the prototype we have built to evaluate our design. The prototype runs at user level and clients communicate with it using *NFS* version 2. All Mammoth file data and metadata are stored as files in an unmodified local file systems of the nodes that run the Mammoth server. Currently, Mammoth runs under Linux and Solaris 2.x; apart from a handful of operating system specific calls that Mammoth uses, it is portable to any system that supports the POSIX interface.

Mammoth clients can be unmodified *NFS* clients, but we have also implemented a modified *NFS* client for Linux 2.4.x and FreeBSD 4.x that augments the standard *NFS* protocol with a *close* operation. Mammoth needs to track file *open* and *close* to determine when to create a new version of the file. Mammoth determines that an *open* was called on a file by tracking the *setattr*, *write*, and *read* RPC calls to the *NFS* server. When unmodified clients access a Mammoth server, the server uses a heuristic to guess when a *close* has occurred.

5.1 Metadata

Mammoth metadata is stored as files in a shadow directory similar to that of AFS [13]. This structure is designed to optimize access to the current version of a file, which can be accessed without reading any on-disk Mammoth metadata, as long as the storing node has registered interest in the file and currently holds either a read or write lock.

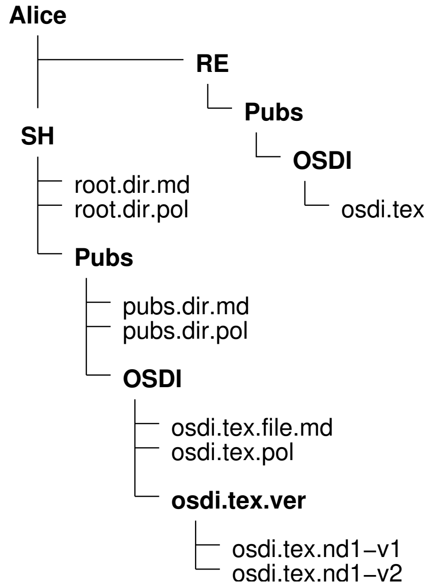

Figure 2: The layout of the file system.

Figure 2 presents an example of a directory tree rooted at `alice` and its associated metadata. Current versions are kept in the `RE` (real) subtree. Metadata and previous versions are stored in the `SH` (shadow) subtree.

When Alice opens `Pubs/OSDI/osdi.tex` the server returns `RE/Pubs/OSDI/osdi.tex`. On a *write* the file `RE/.../osdi.tex` is moved to `SH/.../osdi.tex.nd1-v3` and a new `RE/.../osdi.tex` is created. On a *close* the metadata is updated by appending the new version record to `SH/.../osdi.tex.file.md`.

A directory's metadata consists of two files. The file `dirname.dir.md` stores (1) directory state information and (2) a list of name entries annotated with the time they were created or removed. The file `dirname.dir.pol` contains (1) a list of nodes that are interested in the directory and (2) a list of replica nodes.

A file's metadata consists of two files and a directory. The file `filename.file.md` stores (1) the file's current state: whether it is present or deleted, (2) the file's current and previous owner and lock status, (3) whether the file is up to date, and (4) a list of file version information as described below. The file `filename.file.pol` stores (1) a list of interested nodes, (2) a list of replication nodes, and (3) the file's availability policies and groupings. The replication server list is used by the system replication thread to select nodes to store replicas of this file. There is also a `filename.ver` directory that is associated with every file that stores file versions.

5.2 Naming

As stated previously, directories and files are named internally using globally unique identifiers (GIDs) comprised of their creator node's network address and a local timestamp. Objects are also named by a pathname, just as they are in a traditional UNIX file system. A key difference between Mammoth and such a system, however, is that nodes store arbitrary portions of the file system, not the entire thing. It is thus possible for a node to store a directory without storing its parent or children. This feature creates a problem for path-based naming.

A key advantage of path-based naming is that index structures used to locate objects are hierarchical and thus portions of them can be cached to optimize for locality. The alternative of using a flat space of GIDs to locate objects has the disadvantage that the large object lookup table each node must store cannot be segmented in a similar way. The problem with using pathnames in Mammoth, however, is that renaming a directory changes the pathname of every node in the subtree rooted at the renamed directory and these descendent objects may be stored on a different node from their renamed ancestor. As a result, a single rename operation on one node can invalidate the pathnames of a large number of objects on many nodes in the system.

We resolve this problem by using pathnames as a hint to speed lookup, but by also using GIDs to detect stale pathnames. The objects on a node are stored and located using the most-recent pathname known locally to be valid for them. For each pathname, the node also stores a timestamp vector indicating the time that each pathname component is know to be valid.

When a node renames a directory, it sends a rename message that propagates to every node that stores the directory's descendants. These nodes use this message to update their local file system to reflect the new pathname. This update message, however, is only an optimization. If a node does not receive it and thus uses a stale pathname to lookup an object at another node, either because the originator or the receiver has a stale name, the receiver will detect this when it compares the GID of the object it locates, if any, with the the GID in the request message. A similar procedure is followed to invalidate stale entries in a node's pathname-prefix table.

When a node detects the use of a stale pathname it initiates a lookup starting at the root, using the stale pathname combined with its timestamp vector to locate the desired object. This lookup differs from a standard one only in that now each node searches its directory history for a rename operation involving the pathname component it stores, starting with the component's timestamp. Such an entry, if found, will specify a new pathname and timestamp for the component, which the node uses to continue the lookup. The lookup also assembles a revised pathname and timestamp vector for the file as it goes. When the lookup completes, the target and requesting nodes repair their local file system, if necessary, to reflect the name change. These local file systems are organized such that repairing them for one object automatically corrects the pathname of all other locally-stored objects that are descended from the same renamed directory. This lookup procedure is thus required at most once for each node that stores an affected object and is completely unnecessary when nodes receive the pathname update messages generated when renames occurred.

5.3 Status

The prototype implements a large subset of the design described in this paper. Features that have been fully implemented include: basic file system operations, propagation of metadata updates, interest registration and deregistration, and file locking. Replication is limited to a simple policy that replicates a file when its previous replica has reached a pre-defined threshold age. The design described in Section 3 has been fully implemented with the exception of the step that handles permanent failure of a node, described in Section 3.2.3.

6 Performance

This section reports on the measured performance of our prototype. Our main goal is to ensure that the key operation overheads are reasonable and scalable. Our loose definition of reasonable is: within a factor of 2 of NFS; we expect some slowdown due to the fact the Mammoth is implemented at user level. Section 6.1 reports a number of measurements that reassure us on this front. We also briefly discuss a second performance consideration, the question of how file-grain operations may manifest in inter-node network communication when significant events network partitioning and recovery occur, in Section 6.2.

6.1 File-grained performance

In this section we present several benchmarks to demonstrate that Mammoth's performance is reasonable; we are comparing to standard NFS. Our experimental setup consisted of Pentium IV PCs running at 1.6GHz and Pentium II PCs running at 266MHz. The machines had 256MB of memory and were connected by a 100Mb switch ethernet network. The Pentium IVs were used as the servers and the Pentium IIs were used as the clients. All numbers reported here are the median of 1000 trials on otherwise unloaded machines and network. We found that the mean and the median differed by at most 10%.

Table 1 presents measurements for the basic file system operations. We compared Mammoth to the standard Linux in-kernel NFS server. We are comparable for most operations. The main discrepancies arises in the *open* and the *mkdir* operations. This is where we incur the cost for storing our metadata as regular files. During an *open* we open and read or create the metadata. If the file was opened with `O_TRUNC` we also version at this point. In *mkdir* we create the directory in both the real and the shadow directories, plus instantiate all the metadata. Since we are dealing with a stateless protocol the timings for the reads and the writes is the total elapsed time from the *open* call to the *close* call.

Operation	**Mammoth (μs)**	**NFS (μs)**
create	660	2250
open	6110	1850
read - 1-KB	8430	2000
read - 64-KB	15600	9430
write - 1-KB	8860	3100
write - 64-KB	15300	11900
remove	3700	2900
mkdir	6000	1900
rmdir	3800	1880

Table 1: Timings for *create*, *open*, *read*, *write*, *remove*, *mkdir*, and *rmdir* operations.

We also measured the overhead for several common Mammoth operations. Several of these operations are constantly used to maintain consistency within the system. Table 2 presents the overheads for lock acquisition and invalidation, versioning, propagating metadata, and servicing remote requests for metadata. For lock invalidation and propagating metadata the cost of sending the data to the interested nodes is not included. We measured the cost on both the local and the remote nodes.

Operation	Local Node (μs)	Remote Node (μs)
Lock acquisition	6	2140
Lock invalidation	1230	3640
Update propagation	61	4570
Update request	5160	1190
Versioning - Rename	1590	N/A
Versioning - Copy	6400+161/4KB	N/A

Table 2: Overheads for common Mammoth operations.

6.1.1 Opening the Current Version of a File

To open the current version of a file, a Mammoth node must first acquire a read or a write lock. If the node is the owner of the file then it implicitly holds the write lock. If the file is opened for writing the node must first invalidate all outstanding read locks. This operation requires 1230μs on the owner node and 3640μs for nodes that are currently holding a read lock.

A node that is not the owner of the file must first register a lock with the owner. A requesting node will typically know the owner of the file because ownership changes are eagerly propagated. If the file is opened for reading and the node already has a read lock then there is no additional overhead. Otherwise, the node must register a read lock. The cost to register a read lock is 6μs for the node and 2140μs for the owner node. If the node does not hold the current version there is an additional cost of 5160μs for the local node and 1190μs for the remote node to request the latest version; this cost does not include sending the data;

To open the file for writing requires that ownership of the file be transfered; owning a file implies holding a write lock. The costs of transferring ownership are similar to the cost of registering a read lock. There are two additional pieces of overhead. First, the owner must invalidate all the read locks currently being held. Second, the owner must send out an update to all the interested nodes informing them that the ownership of the file has changed. Invalidating the read locks costs 1230μs on the owner node and 3640μs for nodes that are holding read locks. To inform the interested nodes of an owner changes costs 61μs on the owner side and 4570μs on the read lock holder's side.

When opening a file, locally or remotely, there is a 6110μs overhead to creating the file handle for the file. Once the file handle is created and inserted into the file handle cache this 6110μs overhead is no longer incurred. The file handle stays in the cache as long as the file is in use. The file handle is evicted after a time of inactivity; this length of time is user configurable. The measurements for reading and writing were done with a cold system cache and a cold file handle cache.

6.1.2 Reading

Reading the current version of a file should have roughly the same performance as NFS. To measure read performance we read a 1-KB and a 64-KB file. We measured the total elapsed time for opening, reading, and closing the file. In Mammoth it took 8430μs to read the 1-KB file and 15600μs to read the 64-KB file; it took 2000μs and 9430μs respectively to read a 1-KB and a 64-KB file in NFS. One can see that our reading performance is on par with that of NFS if we account for the 6ms additional overhead for opening the file for the first time.

6.1.3 Writing

Writing to a Mammoth file has extra overhead, compared to NFS, because it requires the creation of a new version of the file. In this experiment we wrote to two different files, one small and one large. In each experiment, we show the elapsed time for creating the new version, performing the described write operation, closing the file, and performing *sync*, to synchronously write the modified data to the server.

To begin, we measure the write performance without versioning. Those results are presented in Table 1. The overhead is similar to that for reading. Next, we *truncate* a small and a large file with versioning turned on. We measured 13900μs for the 1-KB file and 21800μs for the 64-KB file, compared to 2650μs and 11660μs for NFS respectively. Since *truncate* completely erases the contents of a file we are able to use *rename* to version the file. The overhead for doing so is 1590μs. There is also additional overhead in re-creating the file.

We then computed the time for writing various numbers of bytes to each file. Writing 1024 bytes requires 19500μs for Mammoth and 2550μs for NFS, while a larger write of 65536 bytes takes 34300μs in Mammoth and 11200μs in NFS. Since we are updating the file we are forced to copy it to create the new version. This versioning technique gives us a constant overhead of 6400μs with an additional overhead of 121μs per-4KB write. Copying the file to version it adds significant overhead to writing. To reduce this overhead we are working on a copy-on-write mechanism so that we do not have to copy files in their entirety.

6.1.4 Andrew file system benchmark

We ran the Modified Andrew file system benchmark to determine Mammoth's performance in a common operating environment. The Andrew benchmark induces a load similar to that of a developer. When we ran Andrew on Mammoth and NFS we found the results to be almost identical; 15.67s for NFS and 15.87s for Mammoth. From this we draw the conclusion that the large overheads shown by the micro-benchmarks are predominantly not felt by the end user.

6.1.5 Propagating updates

A key feature of Mammoth is that the metadata cached at *interested* nodes is updated eagerly whenever it changes. For this approach to work the overhead associated with creating and propagating these updates must not significantly retard the system and the protocol used to propagate the updates must scale to large number of nodes. We first measured the overhead associated with propagating the update and then we evaluated our two phase commit protocol to determine how well it scales as the number of nodes and the number of updates increase.

We measured the overhead associated with creating an update and receiving the update. It takes $61\mu s$ to create an update. This time does not include the time to propagate the update to other interested nodes. To process an update it takes the receiving node $4570\mu s$. To demonstrate that this overhead does not significantly impact the performance of a Mammoth server, more precisely the client response time, we ran a Mammoth server with 1 to 16 interested nodes and flooded the server with 64-KB write requests. There was no change in the server response time as we added more interested nodes.

6.1.6 The propagation protocol

A well designed propagation protocol must not only reliably deliver updates but also handle node failures without putting additional load on the system. This is especially important in a system like Mammoth because short term transient failures are the norm rather than the exception and thus must be handled efficiently. Existing protocols are either too heavy or do not sufficiently handle the possible large number of failures.

Mammoth requires a protocol that (1) efficiently delivers updates to nodes that are operational, (2) handles delivery failure gracefully without adding additional load to the system, and (3) assuming that the delivery failures are not permanent, eventually deliver the updates to the required nodes. Existing protocols such as TCP and UDP are not sufficient because they are not able to cover the second and third requirement. Traditional two phase commit protocols are too heavy. Due to the relatively large number of expected failures these protocols would not allow progress and put a significant load onto the system. The prototype follows the designed described in Section 3.2.

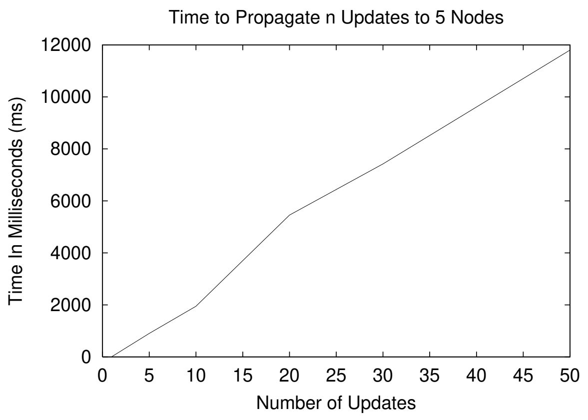

Figure 3: The propagation time of an update as the number of updates increase.

We measured scalability of the propagation protocol using a test harness that simulated a Mammoth server. To simulate large numbers of nodes, multiple instances of the testing harness were run on single machine. The simulation cluster consisted of Pentium II PCs with 256MB of memory connected by a 100Mb switched ethernet network.

We first measure the performance of our propagation protocol when there are no failures. To determine how the protocol deals with an increasing number of updates we measures the time it took to propagate 1 to 50 updates to five nodes. Figure 3 shows the response of the protocol as the number of updates increases; the five nodes experience no faults. We see that the propagation time linearly increases as the number of messages.

Next we investigate the performance of the protocol as we increase the number of nodes. We measure this aspect in terms of the number of messages sent. The underlying delivery mechanism uses UDP. Thus, as the system load increase so does the likelihood that the system is going to drop the packet, either at the sender or at the receiver. As we increase the number of nodes we also increase the number of updates that need to be propagated. This is not the same case of increasing the number of updates. If there are 50 updates for one node and the protocol can not propagate an update then it is not going to attempt to

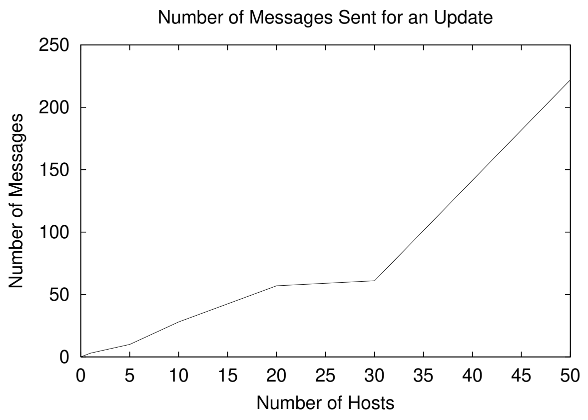

Figure 4: The number of messages sent by the lightweight two phase commit protocol as the number of nodes in the system increase.

propagate the other 49, thus avoiding flooding the system with useless attempts. However, if there is one update for 50 nodes then the system is going to attempt to deliver the updates to all the nodes and in the process flood the underlying delivery transport with messages.

Figure 4 shows the number of messages sent by the underlying delivery transport as we increase the number of nodes that an update is sent to. We see that the number of messages increases nearly linearly, two messages for one update, until we hit about 30 nodes; an update requires two messages, one for each phase. As we increase the number of nodes past 30 we see that the number of messages required increases. At 50 nodes about 200 messages were required, or each messages was resent on average once, as the additional messages cause the underlying transport to drop additional packets.

6.2 Global performance

When a significant system event such as node or network failure or recovery occurs in the system, the actions of the individual nodes concerning individual files as described in this paper appear to be both necessary for correctness and reasonable in cost. However, the number of nodes and files that may be impacted by a single system event may be very large.

To understand the impact of significant system events it is necessary to understand the topology of the system; the connections and dependence among nodes. In Mammoth these connections are defined by the interest and replication sets of directories and files. From the outset Mammoth has been designed to instantiate these connections as required. Initially a node starts as a singly-connected component, and as requests from clients are processed, additional connections are established. This growth, however, is controlled by the replication-set and interest-set formation policies described in Section 4.

Given that Mammoth is a new system, we have not had a chance to deploy it widely, and thus we can really only speculate about the system topology that will actually form. We believe, however, that Mammoth nodes will tend to form cliques that mostly store the data of a related population. This clustering can easily be achieved for replication nodes by controlling the way that replication sets are created, as described in Section 4. The situation is a bit less clear for interest sets, but if most modifiable files are narrowly shared or if sharing tends to be localized to sub-populations, cliques will form. It may also be necessary to reject certain interest-set formation requests, if they tend to form inter-clique connections. A full analysis of this issue is reserved for future work.

7 Related work

Recently, a large body of work has been done in the area of peer-to-peer storage systems. Systems like Gnutella [8] and Napster [14] were devised for the primary reason of sharing information. While other systems such as FreeNet [5] and Eternity [1] were designed to function as deep archival repositories. CFS [6, 21] is a read-only peer-to-peer file system that operates at the block level. To improve the latency associated with the $O(\log N)$ lookup cost, blocks are cached on nodes that are on the path to node that stores the primary copy. PAST [17, 18] was designed to be a general purpose replicated object store.

Ficus's [16] and Mammoth's view differs on the amount of entropy that exists in a file system. Ficus partitions the file system into volumes and a set of replica nodes are assigned to replicate it. These replica nodes are responsible for serving the data and thus there is an implicit assumption that the replica set changes infrequently. Mammoth attempts to adapt to client usage patterns and migrates the data to best serve the client. Like Mammoth, Ficus uses optimistic replication, replica nodes are informed of changes and they lazily pull the updates from the replica node where the data was modified. Instead of using a protocol to ensure replica nodes are informed of new updates, Ficus uses per file version vectors that are propagated by a gossip protocol [2]; this ensures that consistency is eventually reached.

Locus [24] and Coda [10, 12] use tightly-coupled forms of optimistic replication. In Coda, for example a tightly-coupled collection of server nodes replicate a portion of the file system. Connected to this are clients that actively cache file data and depend on the servers for synchronization and consistency. Disconnected clients can modify locally cached objects; these changes are reconciled with the servers when clients reconnect. Mammoth differs in that it avoids tight coupling and that it uses version histories to simplify consistency and conflict reconciliation. In contrast, Coda resolves partitioned updates as part of the partition reconciliation process, marking files with conflicts as unusable until fully reconciled by a user or application.

Mammoth's division of reconciliation responsibilities between the low-level storage system — Mammoth — and higher-level reconciliation, contrasts with Bayou [15] and OceanStore [11]. Bayou supports full application-aware reconciliation integrated with a relational database. It uses operation logging to resolve conflicting updates by merging the logs from conflicting nodes, rolling back the database, and replaying the merged log. OceanStore extends this basic idea for very wide-scale storage.

Echo [4] and JetFile [9] rely on non-standard network transports for their communication. Echo relies on Isis [3], which is a reliable, tightly coupled, distributed communication system. JetFile relies on the availability of multicast. Mammoth does not rely on any special communication primitives, it can use any available transport.

Porcupine [19] is file system designed to support internet services like electronic mail. The consistency mechanisms needed to support such services are significantly simpler because issues with respect to update order do not exist.

Finally, Mammoth's use of versions to simplify replication was inspired by the Cedar[7] file system, which used versioning to simplify cache consistency. In Cedar, each version was named by a unique serial number assigned when it was created. In Mammoth, on the other hand, versions are named by any timestamp contained in the interval between its creation and either the creation of the next version or deletion of the file.

8 Conclusion

This paper has described the design and implementation of Mammoth, a peer-to-peer file system that provides clients with a traditional UNIX-like API while also providing the scalability and grass-roots sharing benefits exhibited by peer-to-peer storage systems such as CFS and PAST.

A Mammoth file system consists of a set of nodes that cooperate to implement a hierarchical file system. Each node can store an arbitrary collection of directories, file metadata and file contents. The only thing that links these nodes together are network addresses stored within these system objects. Each directory or file metadata object encodes the addresses of the nodes that store it. File data itself is stored as a journal of immutable versions, thus freeing replicated file data from consistency issues. A file's metadata lists these versions along with the network address of nodes that store them. As a result, a node can store a directory without storing its parent or children. Similarly, a node can store a file without storing its contents, or alternatively it can store some, but not all of the file's versions. The system can thus scale to a very large number of nodes, as long as each directory or file is stored on only a few of them.

A key issue for any system comprised of many nodes is dealing with failure. Mammoth is designed to be robust in the face of failure. It automatically replicates data to ensure availability and it follows an optimistic scheme that allows nodes to read and write whatever version of data they can currently access. In the absence of failure, one of the nodes that stores a directory or file acts as an owner to coordinate updates. In the event of failure, nodes are allowed to elect new owners from the replicas they can access.

Eventual consistency is achieved by ensuring that the nodes that store a directory or file metadata object eventually receive all updates made to it. The order that these updates are delivered is unimportant, however, because metadata is stored by timestamp operation logs. Update delivery is ensured by enqueueing an update on the node where it occurs until it can be delivered to every node that stores the object. The update is also enqueued by the nodes that receive it until they receive subsequent confirmation from that node that the update has been fully propagated. Until then, these nodes monitor the liveliness of the updater and accept responsibility for propagating the update should that node fail.

We have implemented Mammoth as a user-level NFS server. File metadata is stored in a shadow file system on each of the nodes that run a Mammoth process. The prototype currently runs on Linux and Solaris, and is easily portable to other POSIX-compliant platforms. Our performance measurements indicate no show stoppers, at least for modest size systems, with performance comparable to a typical NFS server in most cases. A key question, of course, is how will the system perform in the massive scale it is intended for. We have shown that none of the key operations have performance that increases with system size. We have also argued that administrative policies that guide the caching and replication decisions the system makes seem likely to preserve this scalability as the system grows. Empirical analysis at this scale is difficult, however, and thus some questions do remain for future work. To this end, we are preparing a publicly available prototype for release.

References

[1] R. Anderson. The eternity service, 1996.

[2] B. Baker and R. Shostak. Gossips and telephones. *Discrete Mathematics*, 2(3):191–193, 1972.

[3] Kenneth P. Birman. Replication and fault-tolerance in the isis system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 79–86, December 1985.

[4] A. D. Birrell, A. Hisgen, C. Jerian, T. Mann, and G. Swart. The Echo distributed file system. Technical Report 111, Digital Equipment Corporation, Palo Alto, CA, USA, October 1993.

[5] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 46–66, 2000.

[6] Frank Dabek, M. Frans Kasshoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 202–215, October 2001.

[7] D. K. Gifford, R. M. Needham, and M. D. Schroeder. The Cedar file system. *Communications of the ACM*, 31(3):288–298, March 1988.

[8] Gnutella. http://www.gnutella.com.

[9] Bjorn Gronvall, Assar Westerlund, and Stephen Pink. The design of a multicast-based distributed file system. In *Operating Systems Design and Implementation*, pages 251–264, 1999.

[10] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 213–25, October 1991.

[11] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*, November 2000.

[12] Puneet Kumar and Mahadev Satyanarayanan. Log-based directory resolution in the Coda file system. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, pages 202–213, 1993.

[13] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. Rosenthal, and F. D. Smith. Andrew: A distributed personal computing environment. *Communications of the ACM*, 29(3):184–201, March 1986.

[14] Napster. http://www.napster.com.

[15] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *Sixteenth ACM Symposium on Operating Systems Principles*, 1997.

[16] Peter L. Reiher, John S. Heidemann, David Ratner, Gregory Skinner, and Gerald J. Popek. Resolving file conflicts in the ficus file system. In *Proceedings of the Summer 1994 USENIX Conference*, pages 183–195, 1994.

[17] Antony Rowstron and Peter Drushel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms, Middleware 2001*, pages ??–??, November 2001.

[18] Antony Rowstron and Peter Drushel. Storage management and caching in past, a large-scale, persistent peer-to-peer stroage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 188–201, October 2001.

[19] Yasushi Saito, Brian N. Bershad, and Henry M. Levy. Manageability, availability and performance in porcupine: A highly scalable, cluster-based mail service. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 1–15, December 1999.

[20] M. Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. Lightweight recoverable virtual memory. *ACM Transactions on Computer Systems*, 12(1):33–57, 1994.

[21] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. Technical

Report TR-819, Massachusetts Institute of Technology, March 2001.

[22] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings Third International Conference on Parallel and Distributed Information Systems*, pages 140–149, September 1994.

[23] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings 15th Symposium on Operating Systems Principles*, pages 172–183, December 1995.

[24] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS distributed operating system. In *Proceedings of the 9th Symposium on Operating Systems Principles, Operating Systems Review*, pages 49–69, October 1983.